SYNTHESIS OF VERIFIABLY HAZARD-FREE ASYNCHRONOUS CONTROL CIRCUITS

by

L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/99

9 November 1990

COVER PAGE

SYNTHESIS OF VERIFIABLY HAZARD-FREE ASYNCHRONOUS CONTROL CIRCUITS

by

L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/99

9 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

SYNTHESIS OF VERIFIABLY HAZARD-FREE ASYNCHRONOUS CONTROL CIRCUITS

by

L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/99

9 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synthesis of verifiably hazard-free asynchronous control circuits*

L. Lavagno
Dept. of EECS
University of California, Berkeley

K. Keutzer
AT&T Bell Laboratories
Murray Hill, NJ

A. Sangiovanni-Vincentelli
Dept. of EECS
University of California, Berkeley

November 9, 1990

Abstract

A synthesis technique for asynchronous sequential control circuits from a high level specification, the Signal Transition Graph (STG) is described. The synthesis technique is guaranteed to generate hazard-free circuits with the unbounded gate-delay model and the bounded wire-delay model, if the STG is live, safe and has the unique state coding property. A proof that STG persistency is not necessary for hazard-free implementation is given.

1 Introduction

Asynchronous sequential circuit design has always been a controversial topic. In the early years of electronic circuit design, when the size of the circuits was such that a human designer could keep track of the complex timing issues involved, it was a popular design style (see [Ung69] for a thorough review). Then synchronous logic dominated the VLSI era, when the ease of design of clocked circuits overwhelmed the advantages of the asynchronous style. Asynchronous design, still, has always been around, at least in the restricted domain of interfaces to the external world, asynchronous by definition. However it was usually limited to finding a good and reliable way to synchronize signals with the internal clock.

Recently there has been a revival of interest in asynchronous self-timed circuits ([Sei81]) due to their desirable properties:

1. the clock-skew problem, getting worse and worse in synchronous sub-micron designs, disappears completely.
2. system-level latency is no longer dictated by the worst-case delay, but by the average delay. For example, a self-timed adder can signal when the result on its outputs is valid and stable, rather than always wait for the worst delay of the carry chain.

These properties are counterbalanced by a more constrained design procedure and, often, by an increase in area, power consumption and worst-case delay.

The clock period of synchronous circuits must be long enough for the combinational logic outputs to settle. Asynchronous circuits, on the other hand, are by definition sensitive to all signal changes, whether they are intentional (i.e. part of the specification) or not. An example of such unintentional changes, also called *hazards*, are the multiple oscillations of a signal that is supposed to have a single transition.

In this paper we will give a procedure transforming a formal, technology-independent specification, called Signal Transition Graph (introduced by [Chu87]), into a circuit implementation made out of "basic gates" such as *nands*, *nors* and S-R flip-flops. We want to prove that the output of our procedure does not have hazards. In order to do so, we must define what delay model we are going to use for our circuit implementation.

- The *unbounded gate-delay* model ([Ung69]) assumes that wires interconnecting gates have zero delay, and that all paths inside each gate (including flip-flops) have exactly the same delay. It also assumes that no bounds are known on the delay of each gate.

*This work was partially supported by the National Science Foundation under Grant UCB-BS16421 and by AT&T Bell Laboratories

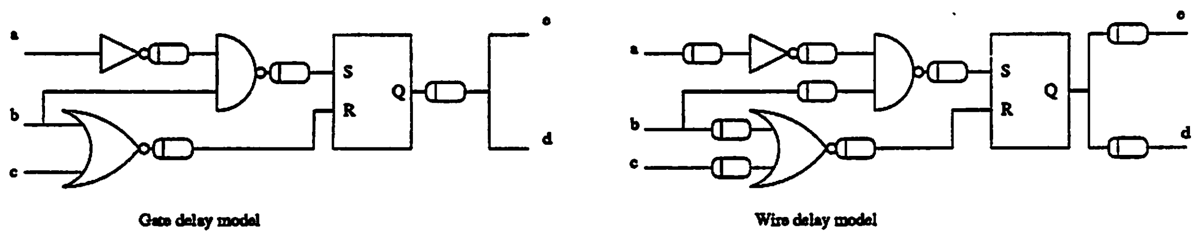

Figure 1: Delay models.

- The *unbounded wire-delay* model assumes that each connection between a gate output and another gate input can have an unbounded delay (see figure 1).
- The *bounded wire-delay* model assumes that each connection between a gate output and another gate input can have a delay. In this model the amount of delay from each input to each output of a complex gate is a function of the load on the gate output. The function depends both on the input that we consider and on the actual circuit used to implement the gate. This delay is called nominal delay. Because of statistical fluctuations in the manufacturing process and of modeling errors, for example the delay on the wires themselves, a lower and an upper bound on the nominal delay are considered when verifying the circuit with timing analysis. This delay model was introduced by [Huf54] (together with the assumption, that we shall not make, that input changes are applied only when the circuit is known to be stable).

In this paper we introduce a new synthesis procedure, that can be proved to generate hazard-free circuits from an STG specification, both with the *unbounded gate-delay* and the *bounded wire-delay* models. The synthesis procedure resembles the one presented in [Chu87] and [Men88]. However our procedure is more general, since it deals with a more realistic delay model, and it can potentially give better results, since it can guarantee that the circuit is hazard-free *without requiring the STG to be persistent* in the case of both unbounded gate-delay and bounded wire-delay. In order to do this we will:

- give a synthesis procedure deriving from an STG a circuit implementation C with two-level combinational functions and flip-flops, together with sufficient conditions on the STG guaranteeing that such an implementation exists,
- characterize all hazards in circuit C,
- show that constrained multi-level logic synthesis can be used without altering hazard properties of C,
- prove that C does not have hazards if we use the unbounded gate-delay model,
- give a procedure to remove all hazards from C if we use the bounded gate delay model,
- show that persistency is not a necessary condition for hazard-free implementation.

The paper is organized as follows. Section 2 recalls some definitions from the literature. Section 3 describes briefly the synthesis procedure introduced by [Chu87] and improved by [Men88] and [Van90]. Section 4 gives a procedure to synthesize a hazard-free circuit under the bounded wire-delay model. Section 5 describes the algorithm implementation and gives experimental results. Section 6 draws some conclusions and outlines some opportunities for future development. Appendix A describes a very simple example, applying the ideas presented in the paper.

2 Definitions

This section gives some basic definitions and previous results useful throughout the paper. Most of the definitions and results, unless otherwise stated, are from [Chu87].

2.1 Logic functions

A *completely specified single-output logic function* g of n input variables is a mapping $g : \{0, 1\}^n \rightarrow \{0, 1\}$. Each input variable x_i corresponds to a coordinate of the domain of g. Each element of $\{0, 1\}^n$ is called a *vertex*.

An *incompletely specified single-output logic function* f of n input variables (called logic function in the following) is a mapping $f : \{0,1\}^n \rightarrow \{0,1,*\}$.

The set of vertices where f evaluates to 1 is called the *on-set* of f, the set of vertices where f evaluates to 0 is called its *off-set*, the set of vertices where f evaluates to $*$ is called its *dc-set*. Each vertex of the on-set of f is called a *minterm*.

A *literal* is either a variable or its complement. A *cube* c is a set of literals, such that if $a \in c$ then $\overline{a} \notin c$ and vice-versa. It is interpreted as the Boolean product of its elements. The cubes with n literals are in one-to-one correspondence with the vertices of $\{0,1\}^n$.

A cube c_1 covers another cube c_2, denoted $c_2 \sqsubseteq c_1$ if $c_1 \subseteq c_2$, for example $\{a, \overline{b}\} \subseteq \{a, \overline{b}, c\}$, so $a\overline{b}c \sqsubseteq a\overline{b}$. The covering is strict, denoted $c_2 \sqsubset c_1$, if $c_2 \neq c_1$.

The intersection of two cubes c_1 and c_2 is the empty cube if there exists x_i such that $x_i \in c_1$ and $\overline{x_i} \in c_2$, otherwise it is $c_3 = c_1 \cup c_2$. It is called "intersection" because it covers exactly the intersection of the sets of vertices covered by c_1 and c_2. For example the intersection of $\{a, \overline{b}\}$ and $\{a, c\}$ is $\{a, \overline{b}, c\}$.

A cube is called an *implicant* of a logic function f if it covers some minterm of f and it does not cover any off-set vertex of f. An implicant of f is called a *prime* if it is not covered by any other (single) implicant of f.

An *on-set cover* F of a logic function f is a set of cubes such that:

1. each cube of F is an implicant of f,
2. each minterm of f is covered by at least one cube of F.

By analogy we can define an off-set cover R of a logic function f as a set of cubes such that:

1. each cube of R covers only off-set vertices of f.
2. each off-set vertex of f is covered by at least one cube of R.

It should be obvious that, if F is an on-set cover of f and R is an off-set cover of f, then for each $c_1 \in F$, $c_2 \in R$ the intersection of c_1 and c_2 is empty.

A cube c_1 in an on-set cover F of a logic function f can be *expanded against an off-set cover R of f* by removing literals from it while its intersection with each $c_2 \in R$ remains empty. The result of the expansion is not unique (it depends on the removal order), but it is always a *prime implicant* of f.

In the following we shall use "cover" to denote on-set covers. Each cover F corresponds to a *unique completely specified* logic function, denoted by $B(F)$. On the other hand a logic function can have in general *many* covers. A cover is interpreted as the Boolean sum of its elements, so it can also be seen as a two-level sum-of-products implementation of the completely specified function $B(F)$.

A cover F is called a *prime cover* of a function f if all its cubes are prime implicants of f. A cover F is called an *irredundant cover* of a function f if deleting any cube from F causes it to be no longer a cover of f (i.e. if some minterm is no longer covered by any cube of F).

The *cofactor of a cube* c with respect to a literal x_i, denoted by c_{x_i}, is:

- the empty cube (a cube that does not cover any element of $\{0,1\}^n$) if $\overline{x_i} \in c$.
- $c - \{x_i\}$ otherwise.

The *cofactor of a cover* F with respect to a literal x_i, denoted by F_{x_i}, is the set of cubes of F cofactored against x_i. The empty cube can always be deleted from a cover, since it does not cover any vertex.

The cofactor has the following property (Shannon decomposition), for each $1 \leq i \leq n$: $B(F) = x_i \wedge B(F_{x_i}) \vee \overline{x_i} \wedge B(F_{\overline{x_i}})$.

A function f is *monotone increasing* in a variable x_i if
$f(\overline{x_i}, \beta) = 1 \Rightarrow f(x_i, \beta) = 1$ for all $\beta \in \{0,1\}^{n-1}$,
that is if "increasing" the value of x_i from 0 to 1 never "decreases" the value of f from 1 to 0.

A function f is *monotone decreasing* in a variable x_i if
$f(x_i, \beta) = 0 \Rightarrow f(\overline{x_i}, \beta) = 0$ for all $\beta \in \{0,1\}^{n-1}$,
that is if "decreasing" the value of x_i from 1 to 0 never "increases" the value of f from 0 to 1.

A function f is *unate* in a variable x_i if it is either monotone increasing or monotone decreasing in x_i. Otherwise f is *binate* in x_i.

A cover F is unate in a variable x_i if variable x_i appears in only one phase (i.e. either x_i or $\overline{x_i}$) in its cubes. A function that is unate can have non-unate covers, but prime covers of unate functions must be unate. Moreover if F is a cover of f and F is unate then f must be unate.

2.2 Hazards

Synchronous circuits do not have hazard problems: the clock cycle is chosen long enough to insure that every latch input is stable when the clock is pulsed. In the asynchronous case, we must make sure that no signal transition ever happens except when it is specified by the designer, because every transition can be recorded by some other part of the system, and cause it to behave incorrectly.

A *static* hazard is a $0 \to 1 \to 0$ transition (static 0-hazard) or $1 \to 0 \to 1$ transition (static 1-hazard) in any condition where no transition for that signal should be enabled according to the specification.

A *dynamic* hazard is a $0 \to 1 \to 0 \to 1$ (or $1 \to 0 \to 1 \to 0$) transition in any condition where a single positive (or negative) transition for that signal is enabled according to the specification.

Hazards must be absolutely avoided, because they can cause the circuit to malfunction in an unpredictable way (for example in response to a change in operating temperature).

The following Theorem was proved in [Ung69]:

Theorem 2.1 *Let T be a two-level representation of a logic function f. Let M be a multi-level representation of f such that it can be obtained from T using only the associative, distributive and De Morgan laws. Then the circuits corresponding to M and T have precisely the same static hazards.*

That is for each pair of input vectors such that the output of M had a hazard for some assignment of wire delays, there exists some wire delay assignment in T such that the same hazard happens at its output, and vice-versa. On the other hand if some hazard could not happen in M under any wire delay assignment, then it could not happen also in S, and vice-versa.

The relevance of this Theorem is that, if we have a two-level implementation T of a logic function f that does not exhibit hazards for some class of input changes, then we can use multi-level synthesis techniques, constrained to use only the transformations listed above, to obtain a multi-level implementation of f that has the same hazard properties.

2.3 Signal Transition Graph

The *Signal Transition Graph* was introduced by [Chu86] as a specification formalism for asynchronous sequential circuits. It is a natural way to specify asynchronous interface circuits, because the causal relations among the signal transitions can be easily described, and it the concurrency is captured explicitly.

A Petri Net is a triple $N =< P, T, F >$, where P is a set of *places*, T is a set of *transitions* and $F \subseteq (P \times T) \cup (T \times P)$, such that dom($F$)$\cup$range($F$) $= P \cup T$, is the *flow relation*. A place $p \in P$ is a *predecessor* of a transition $t \in T$, and t is a *successor* of p, if $(p, t) \in F$. Conversely, a transition $t \in T$ is a predecessor of a place $p \in P$, and p is a successor of t, if $(t, p) \in F$.

A *free-choice net* (FC net) is a Petri net where if a place p has more than one transition $t_1 \ldots t_n$ as its successors, then p must be the *only* predecessor of $t_1 \ldots t_n$. Such a p is called a *free-choice* place. A *marked graph* (MG) is a Petri net where each place p has exactly one predecessor and one successor transition.

An STG is an *interpreted free-choice Petri net*: transitions of the FC net are *interpreted* as value changes on input/output signals of the specified circuit. *Positive* transitions (labeled with a "+") represent $0 \to 1$ changes, *negative* transitions (labeled with a "-") represent $1 \to 0$ changes. From now on t^* will denote a transition of signal t (i.e. either t^+ or t^-) and $\overline{t^*}$ will denote its complementary transition (i.e. either t^- or t^+). *Input transitions* are those that occur on input signals of the circuit, *output transitions* are those that occur on its output signals.

The conventional graphical representation of an STG (slightly different from the Petri net convention) is a directed graph, where nodes correspond to transitions (denoted by single circles) and places (denoted by double circles), while directed edges represent elements of the flow relation. Directed edges fanning out to a transition represent sequencing constraints either on the circuit to be synthesized (if their fanout is an output transition) or on the environment (if their fanout is an input transition). They specify what set of transitions causes each transition.

Figure 2.a contains an example of an STG without FC places (from [Men88]), where x and y are inputs, z is an output. The edge $x^+ \to y^+$ means that the *environment* guarantees that the rising edge of y *always follows* the rising edge of x. The edges $x^- \to z^-$ and $y^+ \to z^-$ mean that the *circuit to be synthesized* must guarantee that the falling edge of z *always follows* the falling edge of x *and* the rising edge of y.

Figure 2.b contains an example of an STG with two FC places, where l, a, c and e are inputs, d and f are outputs. Informally, it describes a circuit where a is allowed to change its value only when l is at 1, and depending on the value of a, one of two full handshake cycles takes place. So the *function* performed by this circuit depends on an external condition a.

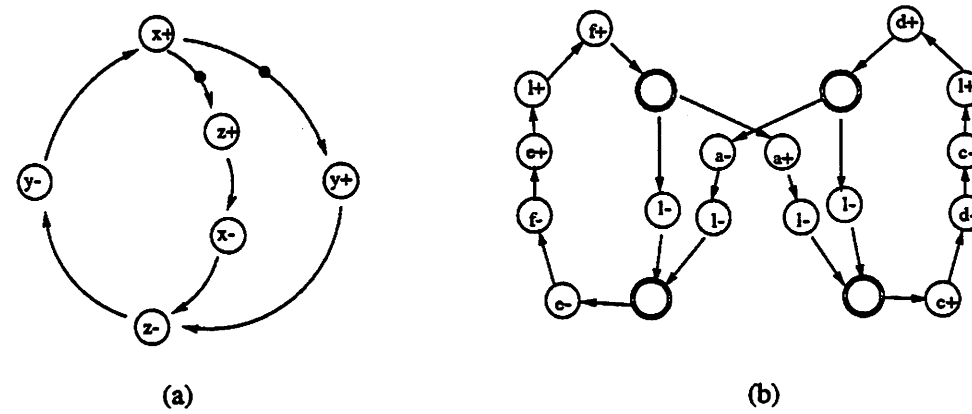

Figure 2: Examples of STG's.

A set T of transitions is called a *complementary set* if it does not contain all the transitions of the STG and $t^* \in T \Rightarrow \overline{t^*} \in T$. A *complementary pair* of transitions is a pair of positive and negative transitions for the same signal (i.e. a complementary set with two elements)..

External signals are those whose behavior is specified by the STG. *Internal signals* are those who are generated by the synthesis procedure to interconnect the logic gates.

2.3.1 Marking and firing

A *token marking* of a Petri net is a non-negative integer labeling of its places. A transition is *enabled* (i.e. the corresponding event can happen in the circuit) whenever all its fanin places[1] are marked with at least one token. Transitions z^+ and y^+ are enabled in Figure 2.a (black dots represent the marking).

An enabled transition must eventually *fire*. This means that the corresponding signal changes value in the circuit. When it fires, a token is removed from every fanin place, and a token is added to every fanout place.

If a place has more than one fanout edge, then exactly one of its fanout transitions is non-deterministically enabled. This means, in practice, that the behavior of the circuit depends on an *external* condition. So we constrain all fanout transitions of an FC place to be *input* transitions.

Two transitions are said to be *concurrent* if there exists some marking where both are enabled. In this case they can fire in any order. Otherwise they are said to be *ordered*.

A set of transitions is said to be *feasible* if it can fire without firing any other transition not belonging to it. For example in Figure 2.a the set $\{x^+, z^+, y^+\}$ is feasible, while the set $\{y^+, y^-\}$ is not, since they cannot both fire without, at least, either x^+ or z^- firing also.

2.3.2 Live and safe net

An FC net is *live* if every transition can be enabled through some (possibly empty) sequence of firings from the initial marking. An immediate consequence of this is that every transition can fire infinitely often. So there is no "dead" part of the net.

An FC net is *safe* if no place can ever be assigned more than one token after any sequence of firings from the initial marking. This means that once a transition has fired, it can fire again only after some other transition has fired (a signal in a circuit cannot rise twice without falling...).

In [Hac72] was proved that a live and safe FC net can be decomposed into

1. FSM components that cover the net (each component is sequential and exhibits non-deterministic choice),
2. MG components that cover the net (each component has concurrency and does not exhibit non-deterministic choice).

"Covering" means that each transition and place of the net has a correspondent in at least one FSM component and a correspondent in at least one MG component.

[1]The marking simply appears on the edges themselves whenever a single fanin/single fanout place is omitted from the graphical representation.

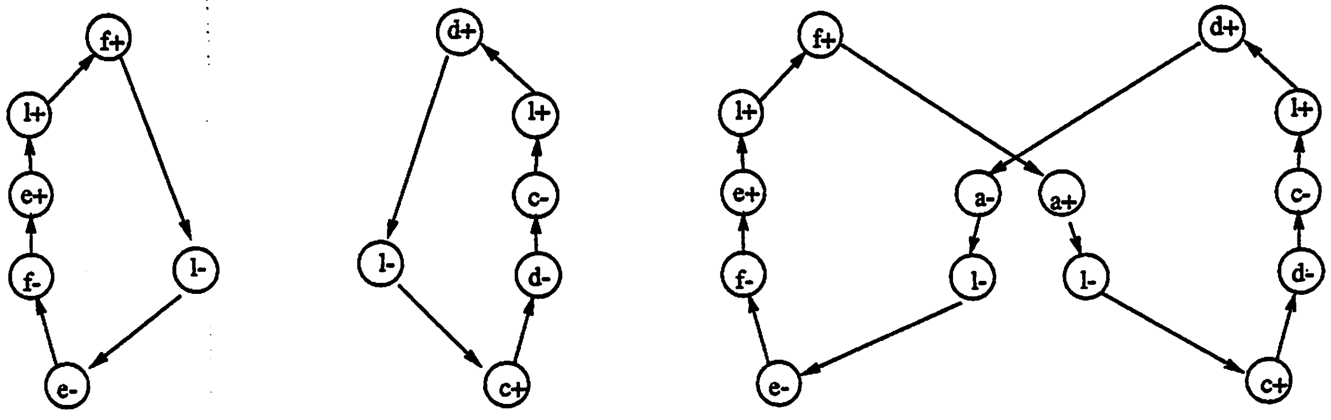

Figure 3: An MG decomposition of the STG in Figure 2.b

FSM components can be thought as running concurrently, synchronizing on the transitions belonging to the intersection of two (or more) components.

MG components can be thought as running one at a time, and whenever a place corresponding to an FC place in the original FC net becomes marked, then the next running component is non-deterministically chosen.

If there are no FC places, i.e. if the net is an MG, then the FSM components are just the simple cycles of the net. For example in Figure 2.a there are two FSM components, namely $x^+ \longrightarrow z^+ \rightarrow x^- \longrightarrow z^- \longrightarrow y^- \rightarrow x^+$ and $x^+ \longrightarrow y^+ \longrightarrow z^- \rightarrow y^- \rightarrow x^+$. On the other hand, Figure 3 contains a possible decomposition of the STG in Figure 2.b in MG's.

This decomposition mechanism is very useful to analyze the properties of the FC net in terms of its components, since those have an easy characterization of behavior properties (liveness, safeness, ...) in terms of syntactic properties.

The following Theorems *about marked graph* components of an FC net are proved in [CHEP71]:

Theorem 2.2 *An MG marking is* live *if and only if the token count in every simple cycle is positive.*

Theorem 2.3 *An MG marking is* safe *if and only if every edge belongs to at least one simple cycle with exactly one token.*

Theorem 2.4 *An MG has at least one live and safe marking if and only if it is strongly connected.*

Theorem 2.5 *A live marking M_1 of a strongly connected MG can produce a marking M_2 (which is also live) if and only if they have the same number of tokens for each simple cycle.*

So all live and safe markings of an MG are partitioned into equivalence classes, where all mutually reachable markings belong to the same class.

The following Theorem is proved in [Chu87]:

Theorem 2.6 *Let N be an FC net such that each FSM component has exactly one token in the initial marking. Then every live and safe marking of N is a live and safe marking of some MG component.*

This Theorem states formally that MG components are "running one at a time".

Liveness is obviously a desirable property of a circuit (a signal that can never change its value is redundant), and safety is required by the synthesis procedure outlined below, so from now on we will restrict ourselves to *strongly connected* STG's with a *live* and *safe* initial marking.

2.3.3 Live STG

An STG is *live*[2] if:

1. it is strongly connected. This ensures that the underlying net is live and safe.
2. for each signal t there is *at least one FSM component*, initially marked with *one token*, such that:

[2]Note the distinction between live *net* and live *STG*.

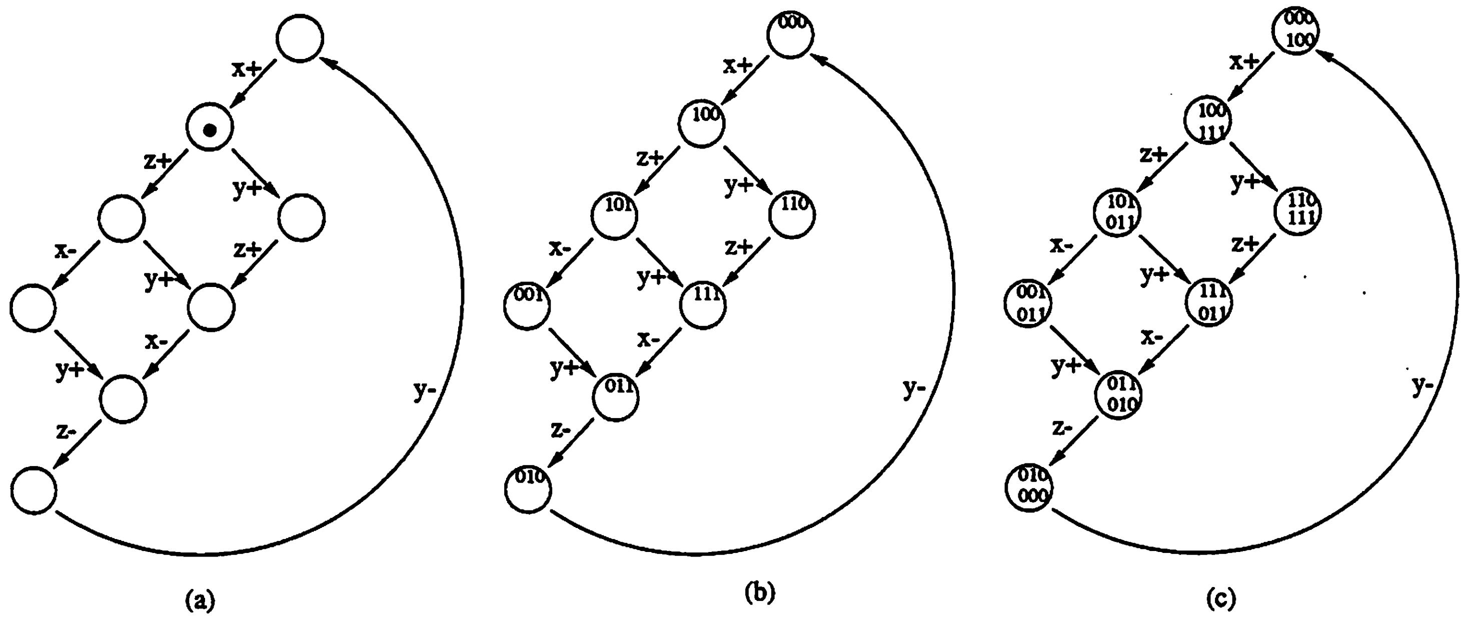

Figure 4: A state graph with state codes

(a) it contains all transitions t^*'s of t,

(b) each path from a transition t^* to another transition t^* (i.e. both rising or falling) contains also the complementary transition $\overline{t^*}$.

This ensures that each signal in the circuit has always a well-defined value in all markings reachable from the initial one, because a rising and falling transition for the same signal can never be concurrently enabled and each signal must have alternate rising and falling transitions.

This definition is broader than the one given in [Chu87], since

- he required that *only two transitions per signal* appear in the STG, and that those transitions are ordered (i.e. belong to a simple cycle) in *every* FSM component of the net.
- we do not restrict the number of transitions per signal, and we require that at least one FSM ensures the alternating order for each signal.

2.4 State Graph

The State Graph (SG) is another possible specification level for an asynchronous circuit, where the concurrency has been been explicitly resolved into a strictly sequential behavior. It can be derived from a live STG using a deterministic procedure ([Chu87]).

The SG is a directed graph, where each node (henceforth called *state*) is in one-to-one correspondence with a live and safe marking of the signal transition graph. An edge joins state s_1 with state s_2 if the corresponding marking M_2 can be reached from M_1 (corresponding to s_1) through the firing of a *single transition*. This transition labels the edge.

Figure 4.a contains the SG derived from the STG in Figure 2.a (the initial marking corresponds to the dotted state).

2.5 Unique state coding

The synthesis procedure described in [Chu87] uses the output signals of the circuit directly as state variables, so the circuit must be able to tell its global state given only the values of the input and output signals.

This means that:

- we must *assign to each state s_i of the SG a unique vector v_i of signal values* and

- this vector of values must be *consistent with the SG edge labeling*, in other words for each edge $s_1 - s_2$:
    1. if it is labeled t^+ then signal t must be 0 in v_1 and 1 in v_2.
    2. if it is labeled t^- then signal t must be 1 in v_1 and 0 in v_2.
    3. otherwise signal t must have the same value in both v_1 and v_2.

An example of such a labeling appears in Figure 4.b.

If this can be done, then we say that the STG from which the SG was derived has the Unique State Coding property (USC, [Van90]).

The following Theorem was proved in [Chu87]:

Theorem 2.7 *An STG S has the USC property if and only if S* is live *and* no complementary set of transitions is feasible in S.

Informally, if a set of transitions is feasible, then they can all fire without any other signal changing in the circuit. If they are also a complementary set, then the initial and final values are the same for all signals, and not all signals of the STG are involved, so we have two distinct states with the same code.

The first formal procedure to enforce the USC property was given by [Van90]. It adds to the STG edges (this reduces the amount of concurrency allowed in the initial specification, so it is not always desirable) and/or new internal signals (i.e. signals used only as state variables and not interesting for the outside world).

3 Logic function derivation from Signal Transition Graph

In the next Sections we will derive an implementation of a live STG with the USC property in two different ways. The first one, taken from the literature, uses the SG as an intermediate step, and it is useful to show that the implementation is valid if the circuit does not have delay. The latter one, which is new, is useful to characterize the hazard properties of the implementation in presence of delays. Both implementation techniques are shown to generate the same result, so either of them can be used to obtain the implementation, but both are useful in order to prove properties of this implementation.

3.1 State Graph derivation from Signal Transition Graph

The SG can be derived from the STG by exhaustive simulation as follows (see also [Chu87] for an equivalent procedure based on graph decomposition):

Procedure 3.1

```
1. for each live and safe marking $M_1$:
   (a) if $M_1$ has not been recorded yet, then:
       i. create a new state $s_1$ associated with $M_1$.
       ii. for each transition $t^*$ enabled in $M_1$:
           A. fire $t^*$, obtaining a marking $M_2$.
           B. call recursively step 1a using $M_2$ as current marking (this call will either create or retrieve the corre-
              sponding state $s_2$).
           C. create an edge from $s_1$ to $s_2$ labeled with $t^*$.
```

3.2 Next-state function derivation from State Graph

Let S be a live STG with the USC property and let n be the number of signals in S. Let f be the next-state/output function to be implemented for signal t[3], and let v_i be an element of the domain of f, $v_i \in \{0, 1\}^n$.

Every state has a unique, consistent encoding in terms of the STG signals, so each SG state s_i can be associated with a vertex v_i of the domain of f.

The following procedure ([Chu87]) derives f:

[3]Each output signal is used as state variable, so the output function is identically equal to the next-state function.

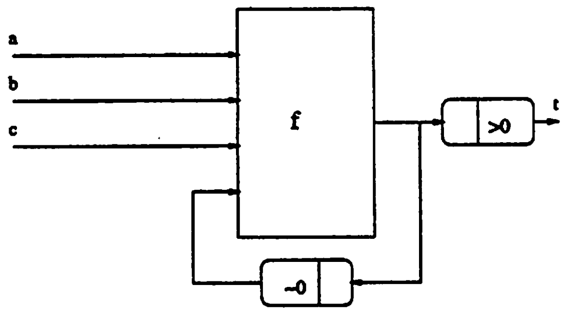

Figure 5: An ideal circuit implementing signal t

Procedure 3.2

1. *for each SG state* s_1*:*
   (a) *let* v_1 *be the corresponding vertex.*
   (b) *if there is no fanout edge* $s_1 \rightarrow s_2$ *labeled* t^* *then let* $f(v_1)$ *be the value of* t *in* v_1.
   (c) *else let* $f(v_1)$ *be the complement of the value of* t *in* v_1.

Notice that in step 1b there can be only one fanout edge with label t^*, because no two transitions for t can be concurrently enabled.

Moreover $f(v)$ is don't care for all vertices v of the domain of f that do not have a corresponding SG state.

Figure 4.c contains the SG corresponding to the STG of Figure 2.a. Each state s_i is labeled with the corresponding input vertex m_i (state code, upper label) and the next-state/output value for x, y, z (lower label).

It should be obvious that a hypothetical implementation of f (see Figure 5) with:

1. zero-delay combinational logic,
2. a feedback loop with almost zero delay and
3. an unbounded delay (greater than the feedback delay) on the output,

would satisfy the STG specification, since a transition on t can happen exactly when it is enabled in the STG (and correspondingly in the SG).

4 Hazard-free logic implementation

4.1 Static hazard analysis of a two-level logic circuit

We want to analyze when *static* hazards can occur in a two-level circuit implementation C_i of a logic function f_i, with the unbounded wire-delay model, given an STG specification restricting the way in which input and output signals of C_i (corresponding to the input variables and the value of f_i) are allowed to change value.

As shown in Section 3.2, we can synthesize an asynchronous sequential circuit specified by an STG as a set of combinational sub-circuits C_i, one for each output signal of the STG t_i, each having as inputs the set of signals specified by the STG. Each C_i, in general, has its output signal t_i appearing also as one of its inputs. In this analysis, though, we will treat them as separate entities, and we will make sure that the output signal does not have hazards if none of the inputs has hazards.

An *input vector* is an assignment of *binary values* to all input signals of C_i. A sequence of input vectors (also called a *transition sequence*) is consistent with the STG if it is a valid firing sequence of the STG. This is equivalent to say that there exists a path on the SG such that each vector appears in it as a state label (in the same order as in the sequence). A static hazard occurs in the circuit if applying a consistent sequence of input vectors, with any delay among them, to C_i, its output changes value when the STG does not allow a transition on it.

Exhaustive simulation of all input vector sequences for all possible delay assignments is clearly not feasible. So we will use *three-valued logic* analysis, where each variable can assume a value of 0, 1 or "-" (for undetermined), as described in [Ung69], by collapsing a whole family of input vector sequences and delay assignments into a single three-valued simulation.

A *controlling value* for an input of a sub-circuit is defined as a value of that input which uniquely determines the value of the output of the sub-circuit, independently of the value of the other inputs of the sub-circuit. This value of the output is the *controlled value* of the sub-circuit. A non-controlling value is any value that is not a controlling value. For example 1 is a controlling value, and 1 is the corresponding controlled value, for any input of an *or* gate, because if it is 1 then the output is 1, while there is no controlling value for any input of an *ex-or* gate.

An *input cube* is a set of assignments of *three-valued values* to all the input signals of C_i. The value "-" is defined as non-controlling for all logic gates.

The three-valued output of a combinational circuit C_i for an input cube c is:

1. the two-valued result if no inputs in c have the value "-".
2. "-" if all inputs to C_i have non-controlling values in c and some inputs in c have the value "-".
3. the controlled value otherwise.

For example the three-valued output of a two-input *or* gate with inputs 1 and "-" (input cube 1-) is 1, while with inputs 0 and "-" (input cube 0-) it is "-".

An input with a value of 1 or 0 corresponds to a signal whose value is *known* to be constant from the STG specification during the transition sequence that we are simulating. An input with a value of "-" corresponds to a signal allowed by the STG to have 1 *or more* transitions during the transition sequence that we are simulating.

With this procedure we can simulate the transition sequence where all the signals with a value of "-" change value in any order at any point in time, under all possible wire delay assignments.

We define an input vector pair (v_1, v_2) to be *valid* with respect to the logic function f_i of signal t_i if

1. $f_i(v_1) = f_i(v_2)$ (since we are analyzing *static* hazards) and
2. state s_2 (corresponding to v_2) is reachable from state s_1 (corresponding to v_1) on the SG without traversing any edge labeled with a transition of t_i.

We forbid valid pairs to include a transition t_i^* because any sub-circuit that implements a transition t_j^* that is enabled by t_i^* must wait for t_i^* to fire. So in order to be sure that t_j^* will be perceived by C_i as *distinct* from the transitions described by the current vector pair, we must assume that C_i plus its feedback loop is ready to accept the next input transition whenever a transition on t_i occurs. We shall see later that this assumption can be reasonably satisfied, implementing the feedback loop with an S-R flip-flop.

Each valid vector pair has associated a *transition cube*, where all signals that change value from v_1 to v_2 are undetermined, while the other signals have the value they have in v_1 (and also in v_2, of course).

It was proved in [Ung69] that a hazard condition exists for a gate-level implementation, with the *unbounded wire-delay* model, if and only if the three-valued simulation of the extended transition cube corresponding to a valid vector pair gives an undetermined output value.

The following procedure performs the three-valued simulation of a two-level circuit, directly implementing an on-set cover F of a logic function f [4], for all valid input vector pairs. Let v^i be the value of input signal t_i in the input vector v, for example if $v = 100$ then $v^1 = 1, v^2 = 0, v^3 = 0$.

Procedure 4.1

```
1. for each valid pair of input vectors $(v_1, v_2)$:
   (a) for each input signal $t_i$:
       i. if any SG path from $v_1$ to $v_2$ includes an edge labeled with a transition for $t_i$, then let $c^i = -$
       ii. else let $c^i = v_1^i$
   (b) i. if $c$ does not intersect any cube $c_i \in F$, then the output of each cube $c_i$, and of the circuit, is 0.
```

[4]That is a circuit with two levels of gates, a level of *and* gates, one for each cube, possibly with *inverters* at their inputs, fanning out to an *or* gate.

ii. *if c covers some cube $c_i \in F$, then the output of that particular c_i, and of the circuit, is 1.*

iii. *otherwise (c intersects some c_i without covering any), the output of those c_i's, and of the circuit, is $-$. Then a hazard condition exists for vector pair (v_1, v_2).*

For example, any sub-circuit synthesized from the STG in Figure 2.a (the corresponding SG appears in Figure 4.b) to implement output signal z must consider the vector pair (010, 110) as valid, with corresponding transition cube $c = --0$ (notice that signal y has two transitions between v_1 and v_2), while the vector pair (101, 010) is not valid, because it requires to traverse the edge labeled z^-.

4.2 Next-state function derivation from Signal Transition Graph

One of the main problems in asynchronous circuit synthesis is to make sure that the circuit behavior is correct for each possible ordering of concurrent transitions. In the example of Figure 2.a, since nothing is said about the ordering of z^+ and y^+, then every output signal must not have static hazards regardless of their firing order. So we must make sure, remembering the analysis in Section 4.1, that the on-set and off-set covers F and R that we synthesize for the next-state function of output t have the following property:

Property 4.1 *Let S be a live STG, let F be an on-set cover and let R be an off-set cover of the next-state/output function of signal t, synthesized from S. Then for each live and safe marking m of S and for each set T of concurrent transitions in S such that t must remain constant during any sequence of firings in T:*

1. *if t must be 1, then there exists at least one cube $c_j \in F$ such that:*
   - *c_j evaluates to 1 in the vertex corresponding to marking m and*
   - *no signal whose transition is in T appears in c_j.*
2. *otherwise, if t must be 0, then there exists at least one cube $c_i \in R$ such that:*
   - *c_i evaluates to 1 in the vertex corresponding to m and*
   - *no signal whose transition is in T appears in c_i.*

We require S to be live in order to be able to associate each marking m of S with a vertex in the domain of the next-state/output function of each signal t in S. We will see later that F and R can have Property 4.1 if and only if S has the USC property.

Case 1 guarantees that the output of F, if so required, remains at 1 independent of the firing order in T. Case 2 guarantees that the output of F, if so required, remains at 0 independent of the firing order in T, even though it is stated in terms of the off-set cover R. This is because the intersection of $c_i \in R$ with any cube $c_j \in F$ is empty. So we can be sure that each $c_j \in F$ will evaluate to 0 in the vertex corresponding to m, and no signal whose transition is in T appears in it.

For example, the set of concurrently enabled transitions in the marking shown in Figure 2.a, corresponding to vertex 100 in Figure 4.b, is $S = \{z^+, y^+\}$. If one of the cubes in the on-set cover of the next-state/output function for z, which must be a constant 1 independent of the firing order in S, is exactly x, then signal z will remain consistently at 1 regardless of the firing order.

The following procedure derives an on-set cover F and an off-set cover R for the next-state function f of signal t_i, receiving as input a live STG, S, having the USC property, with initial marking m. Let v be a vector of values for the n signals that appear in S, $v \in \{0,1\}^n$, and let v_j denote the value of signal t_j in v.

Procedure 4.2

1. *Initialization:*
   (a) *for each signal t_j in the STG, do (determine its initial value):*
      i. *let M_j be an FSM component of S that contains all transitions for t_j, and let m_j be its initial marking (a subset of m).*
      ii. *find on M_j the first transition t_j^* that can be reached from m_j.*

```
    iii. *if* $t_j^*$ *is* $t_j^+$, *then let* $v_j = 0$, *otherwise let* $v_j = 1$.
  (b) *let* $F = \phi$, $R = \phi$.
2. *Recursive step:*
  (a) *if* $t_i^+$ *is enabled in* $m$ *then let* $v_i = 1$.
  (b) *else if* $t_i^-$ *is enabled in* $m$ *then let* $v_i = 0$.
  (c) *for each maximal subset* $T$ *of transitions enabled in marking* $m$ *such that* $t_i^*$ *is* not enabled *in the marking* $m'$
      *obtained from* $m$ *firing all transitions in* $T$ *do:*
    i. *let* $c = \{v_j \; s.t. \; t_j^* \notin T\}$.
    ii. *if* $v_i = 1$ *then let* $F = F \cup c$, *otherwise let* $R = R \cup c$.
  (d) *for each transition* $t_j^*$ *enabled in* $m$ *such that marking* $m'$, *obtained from* $m$ *firing* $t_j^*$, *has not been reached yet,*
      *do:*
    i. *let* $v' = v$.
    ii. *if* $t_j^*$ *is* $t_j^+$, *then let* $v'_j = 1$, *otherwise let* $v'_j = 0$.
    iii. *recursively call step 2 with* $v'$ *and* $m'$.
```

The initial values determined by step 1a are well defined if the STG is live (Section 2.3.3), because:

- at least one M_j containing all transitions of signal t_j must exist.
- the first transition of t^j that can fire on the FSM component M_j starting from a marking m_j must be independent of the path on M_j, or else there would be a firing sequence where two transitions t_j^* (both rising or falling) are not separated by a $\overline{t_j^*}$.
- if there are two (or more) FSM components containing all transitions of t_j, say M_j' and M_j'', then the first transition of t_j reachable from a marking m restricted to M_j' or M_j'' must be the same, since the intersection of the two machines obviously contains all such t^*'s, and the FSM's are synchronized in their mutual intersections.

F and R are constructed by the above procedure exactly to have Property 4.1. This guarantees that the next-state function remains *constant* whenever the STG specifies so, independent of the firing order of a set of concurrently enabled transitions.

Moreover the vector of values v generated at each step is consistent with the firing, so it coincides with the unique code of the state s corresponding to marking m, as described in Section 2.5.

We still want to prove that F and R are on-set and off-set covers of the next-state function f, as obtained in Section 3.

Theorem 4.1 *Let f be the incompletely specified next-state/output function of signal t_i, obtained from a live STG S with the USC property using Procedures 3.1 and 3.2. Let F and R be the covers obtained from S using Procedure 4.2.*

Then every on-set vertex and no off-set vertex of f is covered by a cube of F, and every off-set vertex and no on-set vertex of f is covered by a cube of R (that is F and R are valid on-set and off-set covers of f).

Proof: we have the following cases:

1. Some transitions can fire from the current marking m reaching a marking m' where no transition of t_i is enabled.

   Then we generate a set of cubes such that the vertex v corresponding to m is covered. Moreover all the cubes belong to either the on-set or to the off-set according to whether v_i is 1 or 0, and the decision about v_i is made exactly as in Section 3.2.

   All covered vertices correspond to markings where no transition of t_i can be enabled, so *if the STG has the USC property* (i.e. different markings correspond to different vertices) then no vertex where f must have a different value can be covered.

2. Every transition firing from m will reach a marking m' where a transition of t_i is enabled.

   We will show that in this case no transition of t_i can be enabled in m.

   Suppose that t_i^* is enabled in m and let m' be the marking reached from m firing t_i^*. Then

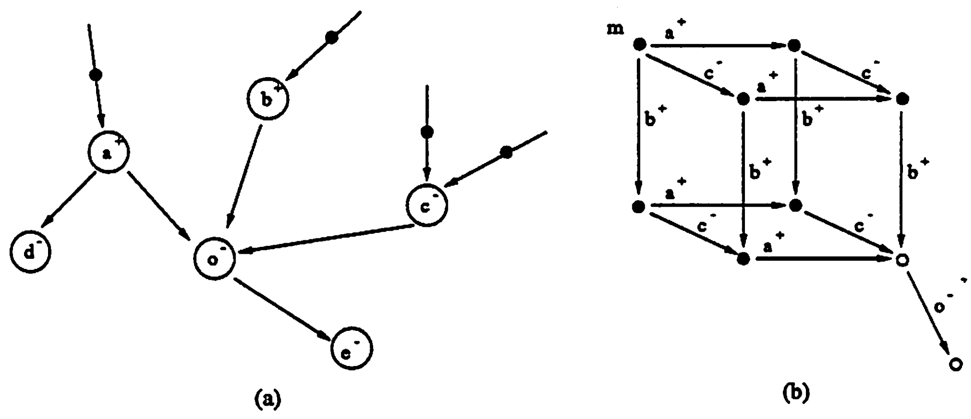

Figure 6: Illustration of Theorem 4.1 case 1

(a) t_i^* could not be enabled again in m', since the STG is live (otherwise two rising or falling transitions of t_i could fire in sequence).

(b) $\overline{t_i^*}$ could not be enabled in m', since the STG has the USC property (otherwise the complementary set $\{t_i^+, t_i^-\}$ would be feasible).

So there would exist some transition (namely t_i^*) that can reach a marking m' where no transition of t_i is enabled, and we have a contradiction.

We know also, since the STG is live, that there exists some marking m'' from which m can be obtained by firing some transition t_j^*. Then whenever the procedure reaches m'', it generates a cube covering also the vertex corresponding to m, because m is obtained from m'' by firing a transition that does not enable t_i^*.

□

Figure 6 contains an STG fragment (a) and the corresponding SG fragment (b) to illustrate case 1. Let o be the signal for which we are generating cover cubes in marking m (black dots in the STG fragment). Black dots in the SG represent on-set vertices of f, white dots represent off-set vertices.

1. The sets of transitions that can fire without enabling o^- are: $S_1 = \{a^+, b^+\}$, $S_2 = \{a^+, c^-\}$ and $S_3 = \{b^+, c^-\}$.
2. The vector corresponding to marking m is: $a = 0, b = 0, c = 1, d = 1, e = 1, o = 1$.
3. The generated cubes are: $c_1 = cdeo$, $c_2 = \bar{b}deo$ and $c_3 = \bar{a}deo$.
4. Each cube covers vertex $\bar{a}\bar{b}cdeo$, corresponding to m, and belongs to the on-set cover. So minterm $\bar{a}\bar{b}cdeo$ of f is covered without problems.
5. Every cube covers only vertices where o^- is not enabled, so no off-set vertex (such as $ab\bar{c}deo$) is improperly covered.

Figure 7 contains an STG fragment (a) and the corresponding SG fragment (b) to illustrate case 2. Let o be the signal for which we are generating cover cubes in marking m (black dots in the STG fragment). The double circles represent an FC place. Black dots in the SG represent on-set vertices of f, white dots represent off-set vertices.

1. The only two transitions that can fire in m are either a^+ or c^- (not both, since this is an FC place, so it enables only one fanout transition). Both enable o^-.
2. One example of a marking m'' predecessor of m is represented as white dots on the STG (replacing the token on $b^+ \to o^-$).
3. One of the cubes generated in m'' is $c_1 = \bar{a}co$, so minterm $\bar{a}bco$ corresponding to m is covered without problems.
4. If one of the enabled transitions in m had been either o^+ or o^-, instead of a^+ or c^-, then it is clear that
   - either the STG would not have had the USC property (o^+ followed by o^-, Figure 7.c),
   - or it would not have been live (o^- followed by o^- Figure 7.d).

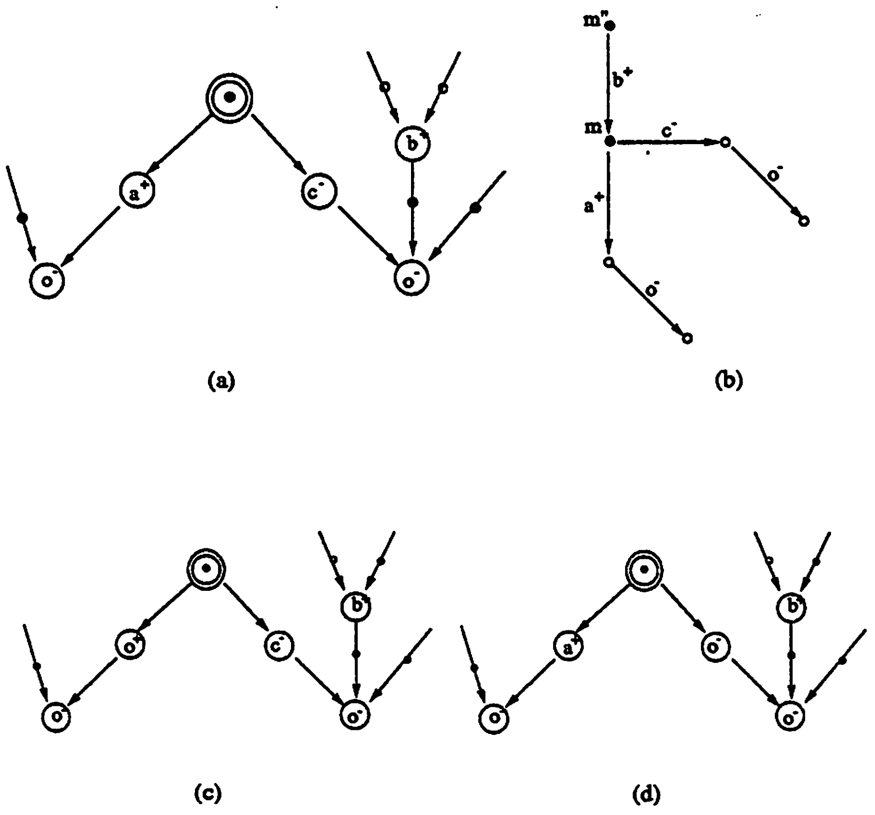

Figure 7: Illustration of Theorem 4.1 case 2

4.3 Circuit implementation of the next-state function

Once we have the on-set and off-set covers of the next-state/output function f for each output signal, we can choose how to implement the feedback loop (sequential part), and apply known logic synthesis techniques in order to obtain a minimal implementation of the combinational part.

4.3.1 Feedback loop implementation with S-M flip-flop

The feedback loop can always be implemented using a simple flip-flop, due to the following theorem, first proved in [Moo90] in the restricted case when the STG is persistent.

Theorem 4.2 *Let S be a live STG with the USC property. Let F and R be a pair of on-set and off-set covers of the next-state/output function f of signal t derived from S according to Procedure 4.2. Let F' be a cover derived from F expanding each cube $c \in F$ against R to a prime implicant of f and removing duplicate cubes.*
Then F' is positive unate *in t.*

An intuitive reason for this is that if f is binate in t, then there is a set of input values for which t oscillates. And, if f is unate, every prime cover of f must be unate.

Proof: let us assume, for the sake of contradiction, that F' is binate or negative unate in t.

Then there exists at least one vertex $v_1 = \bar{t}\beta$, where $\beta \in \{0, 1\}^{n-1}$, belonging to the on-set of f, whose corresponding vertex $v_2 = t\beta$ belongs to the off-set of f (otherwise we could always cover both v_1 and v_2 with a prime implicant not depending on t).

The value of f in vertex v_1 is the complement of the value of t in v_1, so t^+ is enabled in the marking m_1 corresponding to v_1. The marking obtained firing t^+ corresponds exactly to v_2, since v_2 differs from v_1 only in the value of t.

Similarly the value of f in vertex v_2 is the complement of the value of t in v_2, so t^- is enabled in the marking m_2 corresponding to v_2.

m_2 is obtained by m_1 through the firing of t^+, so a firing of t^+ would immediately enable t^-. But this would mean that the complementary set $\{t^+, t^-\}$ is feasible, contradicting the assumption that the STG has the USC property. □

Corollary 4.1 *Let S be a live STG with the USC property. Let f be the next-state/output function of signal t derived from S as described in Section 3.2. Then f is* positive unate *in t.*

Proof: Theorem 4.1 proved that F and R, as derived in Section 4.2, are valid on-set and off-set covers of f, as derived in Section 3.2. Moreover if we expand each cube in an on-set cover to a prime it still remains an on-set cover. And a function with a prime unate cover is unate. □

If f is positive unate in t, then there exist two logic functions s and m that do not depend on t, such that $f = s + tm$. So we can partition the sub-circuit into two purely combinational parts, s and m, and an S-M flip-flop, that is a flip-flop with logic equation $Q = S + QM$ ([BC88]).

The more usual S-R flip-flop has logic equation $Q = S + Q\overline{R}$. We shall see in Section 4.3.2 that the assumption to use S-M flip-flops can be removed without changing the hazard properties of the sub-circuit.

Dynamic hazards are practically unavoidable in any circuit implementation of a combinational function (see [Ung69]). Thus we assume that the S-M flip-flop implementation, or any flip-flop type that we will use, *is relatively immune to dynamic hazards*, i.e. that a dynamic hazard on S or M can cause only one of the following events:

1. Q makes the correct transition in response to the first edge of the hazard.

2. Q goes into a meta-stable state after the first edge, then it makes the correct transition after the second edge[5].

3. Q makes the correct transition after the second edge.

In all cases the correct behavior is guaranteed, only the timing changes.

Moreover we assume that the flip-flop is ready to accept a new transition on its inputs (i.e. that the internal feedback loop is in a stable state) whenever the output Q makes a transition. This means that the internal feedback loop delay must be smaller than the delay on any path from the flip-flop output to one of its inputs. This requirement can always be trivially satisfied by adding buffers after the flip-flop output, with a delay greater than the internal feedback loop delay. These two delays can be guaranteed to be similar (despite process variations, etc.) using appropriate layout techniques, such as keeping the flip-flop and the buffers near to each other.

4.3.2 Logic synthesis for minimal implementation of the combinational logic

In general we have the choice between implementing the on-set or the off-set cover of each signal (inverting the output if necessary). In the following we will discuss only about the on-set cover, but most results apply also to the off-set cover implementation.

We want to obtain an implementation that is minimal with respect to some cost function, usually a combination of delay, area and testability.

Prime and irredundant covers are very important from an implementation point of view, because:

1. heuristic two-level logic minimizers can obtain prime and irredundant covers whose implementation has a nearly minimum area among all two-level implementations of the function ([BHMSV84]).

2. a two-level implementation of a logic function obtained from a prime and irredundant cover is fully testable for single stuck-at faults.

3. a prime and irredundant cover is a good starting point of multi-level logic synthesis systems ([BRSVW87]).

On the other hand we want to preserve Property 4.1, because it is strongly connected with the hazard properties of the implemented circuit.

This means that we can expand each cube in the cover F against R to a *prime implicant* of f, because this does not introduce additional dependencies of the cube on signals that may change when f must remain constant.

Unfortunately we cannot remove *redundant* cubes, unless each cube in the original on-set cover is already covered by some other prime. So we can set up a minimum covering problem similar to the classical Quine-McCluskey minimization

[5]This case is not a problem even if two distinct sub-circuits interpret the meta-stable value of Q differently, since we are assuming bounded wire delays.

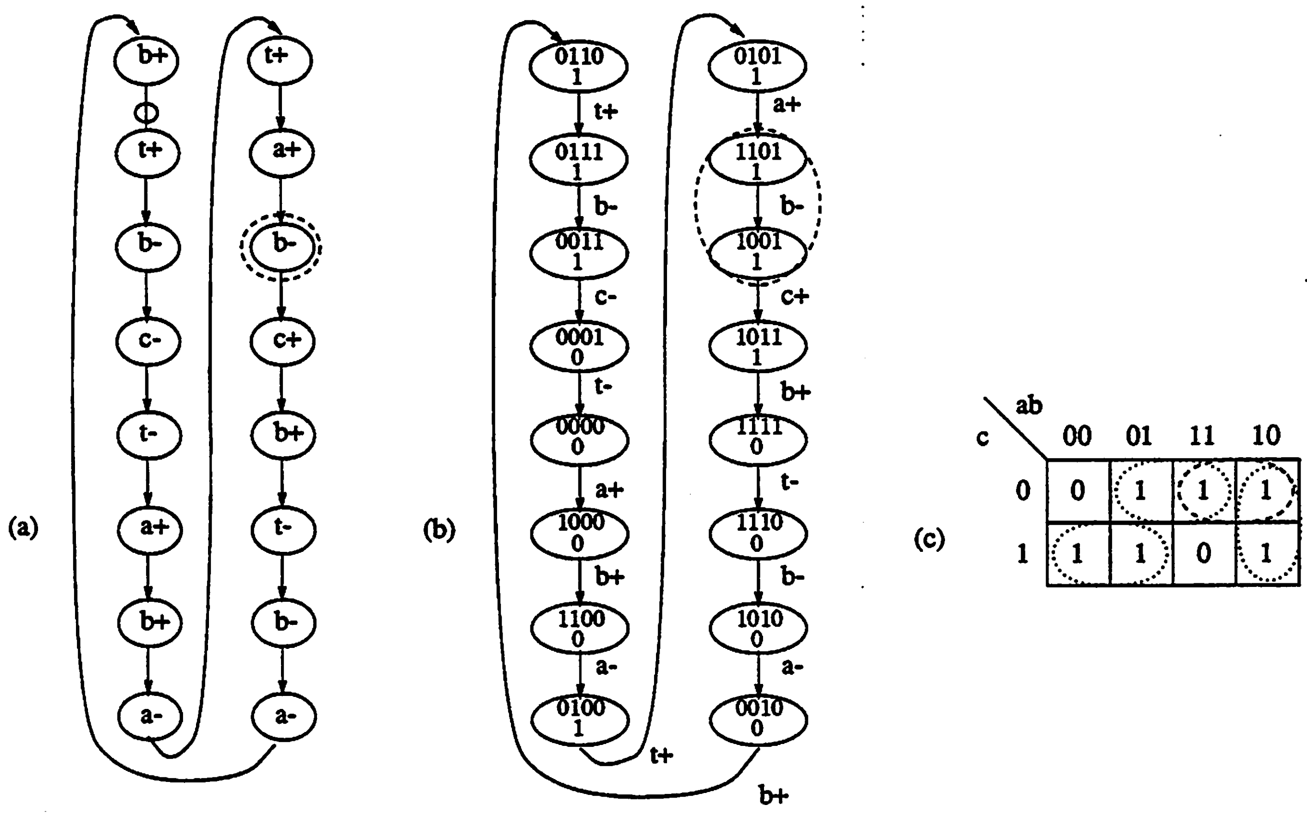

Figure 8: An STG that requires a redundant two-level implementation

procedure[McC56], where each cube c of the original cover (rather than each minterm of f, as in the original procedure) must be completely covered by at least one prime in the output cover.

Figure 8.a contains an example of a live STG with the USC property whose two-level implementation of the flip-flop excitation function m according to Procedure 4.2 is redundant[6]. Figure 8.b contains the SG (input variables are ordered a, b, c, t), while Figure 8.c contains the Karnaugh map of the function $m(a, b, c)$.

The cover of m obtained by the procedure is:
$m = a\bar{b} + a\bar{c} + b\bar{c} + \bar{a}c$
where the implicant $a\bar{c}$ is redundant (it is shown by the dashed oval on the Karnaugh map, while non-redundant implicants are shown by dotted ovals). If the redundant implicant is removed from the cover, then a hazard can occur when the circled b^- transition fires, because the implicant $b\bar{c}$ could go to 0 before the implicant $a\bar{b}$ goes to 1. This causes a static 1-hazard, and possibly a malfunction in the circuit, since the S-M flip-flop can be set incorrectly due to this hazard.

The decomposition into an S-M flip-flop and two combinational networks maintains Property 4.1, since every cube in F has exactly one corresponding cube in the cover of either s or m. Moreover each one of the two covers of s and m is still prime after the decomposition, because we first split F into a cover F_1 where *no cube* depends on t and a cover F_2 where *every cube* depends on the positive phase of t, so cofactoring each cube of F_2 against t leaves it a prime of the cover of m. Theorem 2.1 allows us to use, for example, S-R flip-flops, instead of the less usual S-M type, applying De Morgans' law to the cover of m.

If we want to further improve the area and/or delay performance of the circuit, we can use some multi-level synthesis techniques, such as those described in [BRSVW87]. In order to retain the hazard properties of the two-level circuit, though, we must restrict ourselves to the transformations listed by Theorem 2.1. Algebraic factorization is a direct application of associative and distributive laws, so it can safely be used.

[6]Recall that the cover of m is the set of those cubes in the on-set cover F for signal t that depend on t itself, cofactored against t, while the cover of s is the set of cubes of F that do not depend on t.

4.4 Static hazard analysis of a circuit implemented from an STG

Now let us see what happens when we apply each valid vector pair (v_1, v_2) as described in Section 4.1, to the circuit implementation of signal t obtained from the on-set cover F of the next-state/output function f for t as described in Sections 4.2 and 4.3.

We have the following cases for each transition cube c corresponding to the valid vector pair (v_1, v_2):

1. c is covered by a cube of F or it does not intersect any cube of F: the output is either 1 or 0, and we do not have static hazards.

2. $f(v_1) = f(v_2) = 1$ and two cubes of F, say c_1 and c_2, are required to cover c (the extension to more than two is straightforward): a hazard condition can occur, under an appropriate wire delay assignment.

   Every path on the SG from v_1 to v_2 contains two transitions j^* and i^* (ordered $j^* - i^*$ on the path) such that:

   - i^* turns c_1 off (i.e. c_1 covers the fanin vertex of i^* and it does not cover its fanout vertex on the SG).
   - j^* turns c_2 on (i.e. c_2 does not cover the fanin vertex of j^* and it covers its fanout vertex on the SG).

   We have a hazard if the transition on the output of c_1 propagates to an input of the flip-flop before the transition on the output of c_2, since the STG does not specify any transition for signal t (otherwise (v_1, v_2) would not have been a valid pair).

   Notice that j^* *transitively enables* i^* on the STG, because if they were concurrent, then they would be covered by a single cube.

3. $f(v_1) = f(v_2) = 0$ and c intersects some cube c_1 of F. A hazard condition can occur. under an appropriate wire delay assignment.

   In this case some directed path P in $\{0,1\}^n$ from v_1 to v_2 contains two transitions j^* and i^* (ordered $j^* \rightarrow i^*$ on the path) such that:

   - i^* turns c_1 on,
   - j^* turns c_1 off,

   We have a hazard if i^* propagates to an input of the flip-flop before j^*, since the STG does not specify any transition for signal t. Again j^* *transitively enables* i^* on the STG.

An example of case 2 is the vector pair (000, 110) for the SG in Figure 4.b for output x (see Appendix A). An example of case 3 is the vector pair (001, 010) for the same SG for output x.

Notice also that in case 3 we do not limit ourselves to SG paths, since they cannot traverse an implicant of F, but we must check all possible paths, between v_1 and v_2, where no variable is allowed to change its value more than once.

Both hazard cases happen when the sub-circuit behaves as though *the STG-specified ordering of j^* and i^* was reversed.* This means that the physical circuit implementation must preserve the transition ordering:

Property 4.2 *Given an STG S and a circuit C implementing it, if a transition t_1^* on the input of a sub-circuit C_1 causes a transition t_2^* on its output and the two transitions are ordered in S, then no other sub-circuit C_2 can produce a sequence of events on its output as a delay-free implementation would have produced if t_2^* had preceded t_1^* in time.*

For example in the circuit shown in Figure 9 we must make sure that every path connecting $a - b - c$ through C_1 and C_2 has a longer delay than any path $a \rightarrow c$ in C_2 only.

A suitable global handshaking protocol can guarantee that Property 4.2 is met under the *unbounded wire-delay* model.

For example we can encode each external signal t with two wires, t_0 and t_1, carrying complementary values during quiescent conditions (dual-rail encoding), and driven with wired-or logic. The logic implementation should also follow the guidelines described in [Ung69] for speed-independent circuits, that require disjoint two-level implementation.

Under quiescent conditions all sub-circuits drive one of the wires to 0, say t_0, and leave the other wire at 1. When a transition t^* must occur on signal t, the sub-circuit who generates its transitions drives t_1 to 0 and releases t_0. Then all other sub-circuits, as soon as they recognize the transition, release t_0, that will go to 1 only when all sub-circuits agree that the transition fired. Now any transition enabled by t^* can fire, and so on.

This approach has two main disadvantages:

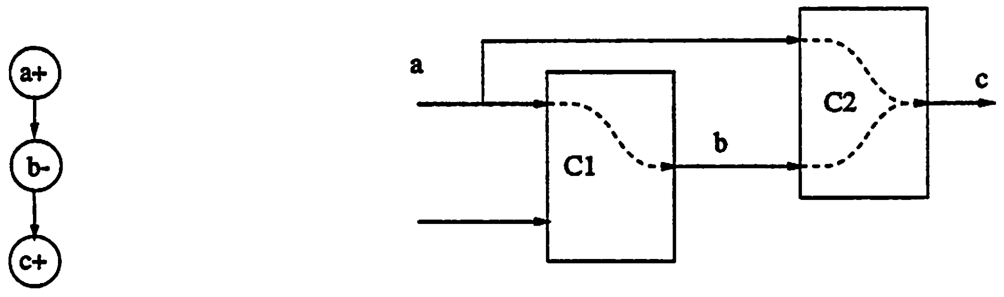

Figure 9: An STG fragment and its implementation.

1. it requires some additional logic (for completion detection and for disjoint two-level implementation).
2. it constrains all the system to follow the speed of the slowest element.

Another way to obtain the same result, under the *bounded gate-wire delay* model, is to *slow down* the output of C_1 so that we are sure that every circuit having a as input is stable when we generate a transition on b.

4.5 Hazard removal procedure

As described in Section 4.4, a hazard can occur when *the difference between the delays along two paths in one sub-circuit is greater than the delay between two transitions.*

So the procedure below identifies all cases when a hazard can occur. The procedure must be called once for each output signal t, with next-state/output function f, implemented by cover F according to Procedure 4.2. Furthermore let s and m be the S-M flip-flop excitation functions as described in Section 4.3.1.

Procedure 4.3

```
1. for each vector pair $(v_1, v_2)$ such that:
     • $f(v_1) = f(v_2)$.
     • $v_1$ and $v_2$ are maximally distant on the SG.
     • no directed path on the SG connecting $v_1$ to $v_2$ contains an edge labeled $t^*$.
   do:
     (a) if $f(v_1) = f(v_2) = 1$ then:
         i. for each directed path $P$ on the SG from $v_1$ to $v_2$, for each pair of distinct implicants $c_1, c_2 \in F$ traversed by $P$:
            A. find the pair of transitions $i^*$ and $j^*$ on $P$ that turn $c_1$ off and $c_2$ on.
            B. let $d_1$ be a lower bound on the delay along the path from input $i$ along cube $c_1$ to $M$ for transition $i^*$.
            C. let $d_2$ be an upper bound on the delay along the path from input $j$ along cube $c_2$ to $M$ for transition $j^*$.
            D. let $d_3$ be a lower bound on the delay between transition $j^*$ and $i^*$.
            E. if $(d_2 - d_1) > d_3$ then a hazard condition exists.
     (b) else ($f(v_1) = f(v_2) = 0$):
         i. for each directed path $P$ in $\{0, 1\}^n$ from $v_1$ to $v_2$, for each implicant $c_1 \in F$ traversed by $P$:
            A. find the pair of transitions $i^*$ and $j^*$ on $P$ that turn $c_1$ on and $c_1$ off.
            B. let $d_1$ be a lower bound on the delay along the path from input $i$ along cube $c_1$ to either $S$ or $M$[7] for transition $i^*$.
```

[7]Depending on whether c_1 belongs to the cover of s or m.

C. let d_2 be an upper bound on the delay along the path from input j along cube c_1 to either S or M^7 for transition j^*.

D. let d_3 be a lower bound on the delay between transition j^* and i^*.

E. if $(d_2 - d_1) > d_3$ then a hazard condition exists.

Notice that the implicants that may cause a hazard in step 1(a)i can only belong to the cover of m, since the SG path from v_1 to v_2 does not use any edge labeled with a transition for t. So all the implicants of f that are involved must contain literal t, because they all are generated by Procedure 4.2 when t has value 1 and no transition for t fires.

The upper and lower bounds on d_1 and d_2 can be obtained by timing simulation on the circuit, where some bounds on the gate delays and wire delays must be known. The input vectors for this timing simulation are just the vectors corresponding to the fanin and fanout states of i^* and j^* on each path. Notice also that we must simulate, except for fanout load effects on the delay, only the part of the circuit corresponding (either directly or through algebraic factorization) to the implicant of F whose delay we are measuring. Otherwise the measurement could be invalidated by a transition on the output of some other implicant.

The lower bound on d_3 can either be obtained in the same way (if we are also synthesizing the circuit for signal i) or from any other information source, such as a data sheet. In the worst case we can assume it to be zero.

Whenever Procedure 4.3 finds a hazard condition, we record the triple of paths and delays.

At the end, we have two choices for each triple (d_1, d_2, d_3) that failed the above test:

1. speed up d_2 and/or slow down d_1. This brings the circuit closer to the gate-delay case, where all delays inside the sub-circuit for each output t are supposed to be balanced. This may not always be possible, furthermore it may introduce cyclic dependencies, so that when we try to remove one hazard we can make another one worse.

2. slow down d_3. This is always possible (just add buffers after the output of the sub-circuit for signal i). Furthermore we are guaranteed that *we do not introduce cyclic constraints*, since d_1 and $d_{,2}$ are measured from a sub-circuit input to a flip-flop input, so adding delay *after* a flip-flop does not change any of them.

If we take the second choice, then we just record by how much each flip-flop output fails to pass the test, and then slow it down by the maximum difference.

Notice the close similarity between what we do here and what is classically done in *synchronous circuit* synthesis:

- in the *synchronous* case we *slow down the clock signal until no more events are propagating along the whole circuit.*

- in the *asynchronous* case we *slow down each signal until no more events that caused its change are propagating along the whole circuit.*

So this approach, even if it does not generate speed-independent or delay-insensitive circuits, can still be considered faithfully adherent to the "asynchronous philosophy", in that every element must obey a "locally defined" protocol, and elements that are logically far apart must not be slowed down due to each other.

If we are synthesizing also the circuit for signal i, a specially bad case for Procedure 4.3 would be $d_3 = 0$, that corresponds to signal i being identically equal to signal j. But signal i can be equal to signal j, given Procedure 4.2, if and only if every transition for j is immediately followed by the same transition for i. Such condition can be easily detected on the STG, and we can just collapse i^* nodes into j^* nodes without changing the specified behavior. In all other cases, the delay between j^* and i^* is at least one logic gate.

4.6 STG Persistency and hazards

An STG is *persistent* ([Chu87]) if all its transitions are persistent. A transition u^* is persistent if for each immediate predecessor t^* of u^*, u^* and $\overline{t^*}$ are ordered.

For example in Figure 2.a transition y^+ is not persistent, because it has x^+ as a predecessor, but x^- and y^+ are concurrent (that is not ordered).

In a persistent STG whenever a transition u^* becomes enabled, none of its enabling signals t can change level before u^* has fired.

A transition t^* *disables* a transition u^* if there exists an STG marking m where both t^* and u^* are enabled, but firing t^* brings to a marking m' where u^* is no longer enabled. Notice that the only transitions that can be disabled in an FC net are FC places, so strictly speaking this is not a characteristic of the specification but of the implementation, where such a disabling can actually occur (we will see below when).

Persistency at the STG level was considered to be a necessary and sufficient condition for the existence of a hazard-free circuit implementation, due to the the following Theorem, taken from [Chu87] (only the notation is changed here, to be consistent with the rest of the paper):

Theorem 4.3 *Let S be a live STG. For each output signal u, there exists a signal t and a marking m in S such that:*

- *t^* and u^* are enabled in m and*
- *t^** disables *u^* and*
- *u^* does not disable t^**[8]

if and only if $\overline{t^*}$ is a predecessor of u^*, and u^* and t^* are concurrent (that is u^* is non-persistent).

The case in which u^* is disabled by t^* but not vice-versa is clearly dangerous, because if the circuit implementing signal u is not "fast enough" to fire after $\overline{t^*}$ fires, then a firing of t^* may prevent u^* from firing. So we could have a potential hazard, depending on the delay of the circuit implementing u and the time from $\overline{t^*}$ to t^*. On the other hand a simple persistency check on the STG would be enough to guarantee that no such hazard occurs.

The proof of this theorem was not based only on STG properties, since strictly speaking no transition in an FC net can be disabled, but also on specific assumptions on the *circuit implementation* derived from the STG, namely that if u^* is enabled by $\overline{t^*}$, then an occurrence of t^* must disable u^* in all circuit implementations of signal u. This is not true in general, but only if all inputs, except t, to the sub-circuit implementing signal u have a non-controlling value in marking m, because otherwise the output of the sub-circuit does not depend on input t.

See for example the STG in Figure 2.a, where y^+ is not persistent. The logic equation for y is $y = x + z$, and using the *gate-delay* model, as assumed by [Chu87], we know that x^- is caused by z^+, so when x^- fires z is already at 1 (a controlling value for the *or* gate), and x^- cannot disable y^+.

Moreover Theorem 4.3 guaranteed hazard-free implementation only if the whole sub-circuit implementing each signal u could be satisfactorily modeled as a *single gate* with the *unbounded gate delay* model. But this model is a reasonable approximation of reality only if the whole sub-circuit was only *one simple gate*, such as a *nand* or a *nor*.

We can examine now the example in Figure 10 (from [Chu87]), where a circuit implementation was derived from a persistent STG (only a fragment is shown here). The value of signals in the given marking is $La = 0$, $Lr = 1$, $Sa = 0$, $Sr = 1$, $Ca = 0$ and $Cr = 0$. When La^+ fires, then the output of a_3 has a rising transition. Suppose that the gate delay of a_3 is greater than the gate delay of a_2 plus the delay between Sa^+ and Sr^-, then a hazard occurs on Lr. The only way to avoid the hazard in the unbounded gate delay model is to assume that the whole group of gates i_1, a_3 and o_1 can be modeled with a *single delay* on the output of o_1.

So we can now state that:

Theorem 4.4 *Let S be a live STG with the USC property. Let C be a circuit implementation of the signals in S according to Procedure 4.2 and the decomposition described in Section 4.3.1. If the implementation of each combinational sub-circuit exciting each flip-flop input, as well as each flip-flop, can be considered single gates with non-zero delay, then C is hazard-free with the* unbounded gate-delay *model.*

Proof: each signal i in S is implemented by a circuit with non-zero delay, so $d_3 > 0$. Moreover both d_1 and d_2 are delays within the same gate, implementing either s or m, so $d_1 = d_2$, and $d_1 - d_2 = 0 < d_3$. □

Notice that the assumption that each excitation function may be modeled as a single gate was used, as shown above, by [Chu87] and [Men88].

5 Experimental results

All the algorithms described in this paper have been implemented within the *sis* sequential logic synthesis system (developed at U.C. Berkeley).

[8]This means that t^* and u^* are not both successors of an FC place, otherwise u^* would disable t^* as well.

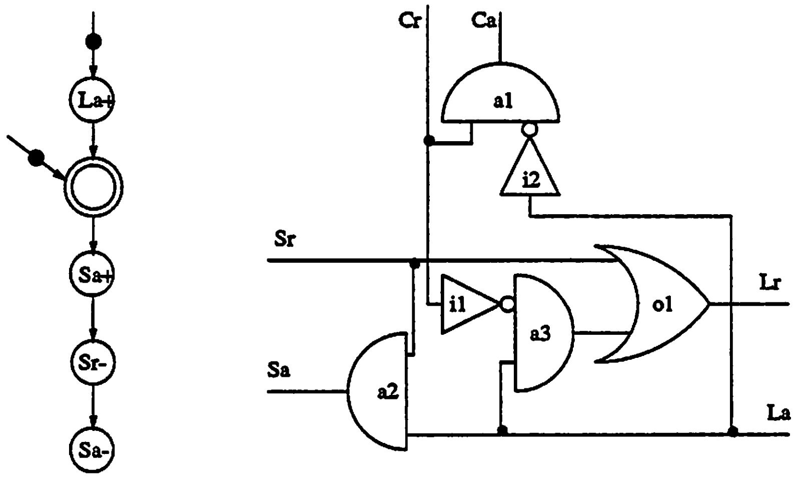

Figure 10: An STG fragment and its implementation

example	Library 1				Library 2			
	With hazards		Without hazards		With hazards		Without hazards	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay
c-elem	50	1.2	50	1.2	96	7.6	96	7.6
ebergen	221	4.4	285	5.2	331	7.8	395	10.4
fifo	89	2.8	89	2.8	112	7.6	112	7.8
luciano	580	7.8	580	7.8	857	17.2	857	17.2
rlm	131	3.0	195	5.2	147	7.8	275	12.6
rlm1	284	4.2	316	5.4	332	9.0	652	21.2
rpdft1	693	8.4	693	8.4	1009	16.0	1009	16.2
test	132	1.6	132	1.6	148	6.6	148	6.6
vbe10b	622	4.6	654	5.2	784	15.4	1072	17.6
vbe5b	221	4.4	253	4.4	245	9.0	309	9.0
vbe5c	161	2.8	193	3.8	153	9.0	249	9.0
xyz	130	3.0	130	3.0	137	7.6	201	7.8

Table 1: Experimental results

Table 1 contains the results of the synthesis and hazard removal procedures for a set of STG examples taken from the literature. The examples were implemented using standard cells, and we used two different libraries, one (Library 1) with a large set of combinational functions and one (Library 2) with only a few gates with widely unbalanced delays.

The columns labeled "area" contain the total area (excluding routing) of each circuit, while the columns labeled "delay" give the maximum combinational logic delay between any two flip-flops or between an external input signal and a flip-flop.

6 Conclusions and future work

The principal target of this paper was to show that each live STG's with the USC property has a hazard-free asynchronous implementation, using both the unbounded gate-delay and the bounded wire-delay models.

In order to prove this, we gave a synthesis procedure, and we examined what were the hazard properties of the result of each step, taking care that we did not introduce new causes of hazards, and we removed old ones at each step.

One important consequence of this work is that persistence can no longer be considered a necessary condition for hazard-free implementation. This is a desirable result, since enforcing persistence reduces the concurrency at the STG level. So we can claim that if the speed measure is the global throughput of the circuit, and if the throughput is bound by the amount of parallelism in the implementation, ours is the *fastest solution* that can be obtained from the given STG specification.

Notice that liveness and USC are only *sufficient* conditions for an STG to have a hazard-free implementation, since our procedure is based on the assumption that only the values of signals specified by the STG can be used as state variables to encode the SG. So in order to give necessary and sufficient conditions for hazard-free implementation, we will have to remove the restriction on the state assignment technique, and state those conditions in terms of properties of both the STG specification and the state assignment procedure.

References

[BC88] C. Berthet and E. Cerny. Synthesis of speed-independent circuits using set-memory elements. In G. Saucier, editor, *Proc. Int'l. Workshop Logic and Arch. Synthesis for Silicon Compilers.* Grenoble, France, May 1988.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[Chu86] T. A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.

[Chu87] T. A. Chu. *Sinthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, June 1987.

[Hac72] M. Hack. Analysis of production schemata by petri nets. Technical Report TR 94, Project MAC, MIT, 1972.

[Huf54] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–190,275–303, March 1954.

[McC56] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, November 1956.

[Men88] T. Meng. *Asynchronous Design for Digital Signal Processing Architectures.* PhD thesis, U.C. Berkeley, November 1988.

[Moo90] C. W. Moon. On synthesizing logic from signal transition graphs. Personal communication, 1990.

[Sei81] C. L. Seitz. *Introduction to VLSI Systems.* Chapter 7, Mead and Conway (Eds.), Addison Wesley, 1981.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[Van90] P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. Submitted to the International Conference on Computer-Aided Design, 1990.

Appendix A A synthesis example

In this section we will go through the synthesis procedure for an example, taken from [Men88]. Its STG, with the initial marking, appears in Figure 2.a, and its SG appears in Figure 4.b.

Let us implement an on-set and an off-set cover for each signal. The initial value vector is $x\overline{y}\overline{z}$.

x :

- only y^+ can fire without enabling x^-.
  We add $x\overline{z}$ to F (since the value of x is 1 in the current value vector).
  1. fire y^+. The value vector becomes $xy\overline{z}$.

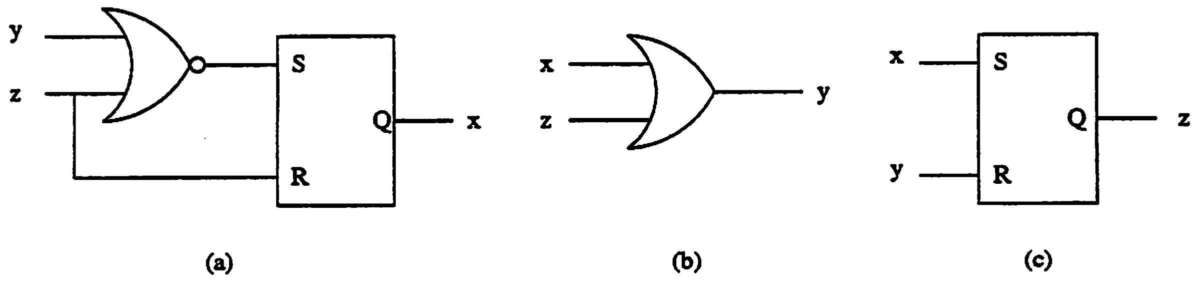

Figure 11: A circuit implementation of the example STG

- z^- can fire. The value vector becomes $\overline{x}y\overline{z}$. We add $\overline{x}y$ to R.
  Fire z^-. the value vector remains $\overline{x}y\overline{z}$.
- y^- can fire. We add $\overline{x}\overline{z}$ to R.
  Fire y^-. The value vector becomes $\overline{x}\overline{y}\overline{z}$.
- Nothing can fire without enabling z^+, so we do not add cubes.
  We reach an old marking, so we return.

2. fire z^+. The value vector remains $x\overline{y}z$.
   - y^+ can fire. We add xz to F.
     x^- can fire. We add $\overline{y}z$ to F.

   (a) Fire x^-. The value vector becomes $\overline{x}\overline{y}z$.
       * Nothing can fire without enabling z^-, so we do not add cubes.
         We reach an old marking, so we return.

   (b) Firing y^+ would reach an old marking, so we return.

So we generate $F = x + xy + xz + \overline{y}z$ and $R = \overline{x}y + \overline{x}\overline{z}$. We can delete covered cubes obtaining $F = x + \overline{y}z$, and the implementation of z described in Figure 11.c.

The following valid vector pairs originate transition cubes that might produce hazard conditions for each signal:

x :

1. valid vector pair $(001, 010)$, with transition cube $0--$. The path $001 - 000 - 010$ traverses the implicant $\overline{y}\overline{z}$. The two transitions that turn it on and off are z^- and y^+ respectively.
   A pair of vectors causing the falling transition on the path from input z through implicant $\overline{y}\overline{z}$ to output s_x is $(\overline{y}z, \overline{y}\overline{z})$. This gives us d_1.
   A pair of vectors causing the rising transition on the path from input y through implicant $\overline{y}\overline{z}$ to output s_x is $(\overline{y}\overline{z}, y\overline{z})$. This gives us d_2.
   d_3 can be measured on the circuit for output z applying transition y^+, that is from vertex $\overline{x}\overline{y}z$ to $\overline{x}yz$.
   d_3 is the reset delay of an S-R flip-flop, so it is generally going to be larger than the difference between the switching times of two inputs of a *nor* gate. So we do not have to slow down the circuit.

All other transition cubes are either covered by implicants or do not intersect them at all.

y : notice that the circuit for output y is purely combinational in this particular case, so $s = f$ and $m = 0$.

1. valid vector pair $(100, 001)$, with transition cube $-0-$. The path $100 - 101 - 001$ traverses the two implicants x and z. The two transitions that turn off implicant x and turn on implicant z in the cover of f_y are x^- and z^+ respectively.
   A pair of vectors causing the falling transition on the path from input x through implicant x (a trivial implicant indeed...) to output f_y is $(x\overline{z}, \overline{x}\overline{z})$. This gives us d_1.
   A pair of vectors causing the rising transition on the path from input z through implicant z to output f_y is $(\overline{x}\overline{z}, \overline{x}z)$. This gives us d_2.

- Nothing can fire without enabling x^-, so we do not add cubes.
  Fire z^+. The value vector becomes xyz.
- x^- can fire. The value vector becomes $\bar{x}yz$. We add yz to R (since the value of x is 0 in the current value vector).
  Fire x^-. The value vector remains $\bar{x}yz$.
- z^- can fire. We add $\bar{x}y$ to R.
  Fire z^-. the value vector becomes $\bar{x}y\bar{z}$.
- Nothing can fire without enabling x^-, so we do not add cubes.
  Fire y^-. The value vector becomes $\bar{x}\bar{y}\bar{z}$.
- x^+ can fire. The value vector becomes $x\bar{y}\bar{z}$. We add $\bar{y}\bar{z}$ to F. We reach an old marking, so we return.

2. fire z^+. The value vector becomes $x\bar{y}z$.
   - x^-, y^+ can fire. The value vector becomes $\bar{x}\bar{y}z$. We add z to R.

   (a) Fire x^-. The value vector remains $\bar{x}\bar{y}z$.
      * y^+ can fire. We add $\bar{x}z$ to R. We reach an old marking, so we return.

   (b) Firing y^+ would reach an old marking, so we return.

So we generate $F = x\bar{z} + \bar{y}\bar{z}$ and $R = yz + \bar{x}y + z + \bar{x}z$. F is already a prime cover, and we obtain an implementation of x as described in Figure 11.a.

y :

- y^+, z^+ can fire. The value vector becomes $xy\bar{z}$.
  We add x to F.

  1. fire y^+. The value vector remains $xy\bar{z}$.
     - z^+ can fire. So we add xy to F. Fire z^+. The value vector becomes xyz.
     - x^- can fire. We add yz to F.
       Fire x^-. The value vector remains $\bar{x}yz$.
     - Nothing can fire without enabling y^-, so we do not add cubes.
       Fire z^-. the value vector becomes $\bar{x}y\bar{z}$.
     - x^- can fire. the value vector becomes $\bar{x}\bar{y}\bar{z}$. We add $\bar{x}\bar{z}$ to R.
       Fire y^-. The value vector remains $\bar{x}\bar{y}\bar{z}$.
     - Nothing can fire without enabling y^+, so we do not add cubes.
       We reach an old marking, so we return.
  2. fire z^+. The value vector becomes xyz.
     - x^-, y^+ can fire. The value vector becomes $\bar{x}yz$. We add z to F.

     (a) Fire x^-. The value vector becomes $\bar{x}yz$.
        * y^+ can fire. We add $\bar{x}z$ to F. We reach an old marking, so we return.

     (b) Firing y^+ would reach an old marking, so we return.

So we generate $F = x + xy + yz + z + \bar{x}z$ and $R = \bar{x}\bar{z}$. We can delete covered cubes obtaining $F = x + z$, and the implementation of y described in Figure 11.b.

z :

- y^+, z^+ can fire. The value vector becomes $x\bar{y}z$. We add x to F.
  1. fire y^+. The value vector becomes xyz.
     - z^+ can fire. We add xy to F. Fire z^+. The value vector remains xyz.
     - Nothing can fire without enabling z^-, so we do not add cubes.
       Fire x^-. The value vector becomes $\bar{x}yz$.

Notice that liveness and USC are only *sufficient* conditions for an STG to have a hazard-free implementation, since our procedure is based on the assumption that only the values of signals specified by the STG can be used as state variables to encode the SG. So in order to give necessary and sufficient conditions for hazard-free implementation, we will have to remove the restriction on the state assignment technique, and state those conditions in terms of properties of both the STG specification and the state assignment procedure.

References

[BC88] C. Berthet and E. Cerny. Synthesis of speed-independent circuits using set-memory elements. In G. Saucier, editor, *Proc. Int'l. Workshop Logic and Arch. Synthesis for Silicon Compilers*. Grenoble, France, May 1988.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[Chu86] T. A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.

[Chu87] T. A. Chu. *Sinthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[Hac72] M. Hack. Analysis of production schemata by petri nets. Technical Report TR 94, Project MAC, MIT, 1972.

[Huf54] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–190,275–303, March 1954.

[McC56] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, November 1956.

[Men88] T. Meng. *Asynchronous Design for Digital Signal Processing Architectures*. PhD thesis, U.C. Berkeley, November 1988.

[Moo90] C. W. Moon. On synthesizing logic from signal transition graphs. Personal communication, 1990.

[Sei81] C. L. Seitz. *Introduction to VLSI Systems*. Chapter 7, Mead and Conway (Eds.), Addison Wesley, 1981.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.

[Van90] P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. Submitted to the International Conference on Computer-Aided Design, 1990.

Appendix A A synthesis example

In this section we will go through the synthesis procedure for an example, taken from [Men88]. Its STG, with the initial marking, appears in Figure 2.a, and its SG appears in Figure 4.b.

Let us implement an on-set and an off-set cover for each signal. The initial value vector is $x\overline{y}\overline{z}$.

x :

- only y^+ can fire without enabling x^-.
  We add $x\overline{z}$ to F (since the value of x is 1 in the current value vector).
  1. fire y^+. The value vector becomes $xy\overline{z}$.

C. *let d_2 be an upper bound on the delay along the path from input j along cube c_1 to either S or $M^?$for transition j^*.*

D. *let d_3 be a lower bound on the delay between transition j^* and i^*.*

E. *if $(d_2 - d_1) > d_3$ then a hazard condition exists.*

Notice that the implicants that may cause a hazard in step 1(a)i can only belong to the cover of m, since the SG path from v_1 to v_2 does not use any edge labeled with a transition for t. So all the implicants of f that are involved must contain literal t, because they all are generated by Procedure 4.2 when t has value 1 and no transition for t fires.

The upper and lower bounds on d_1 and d_2 can be obtained by timing simulation on the circuit, where some bounds on the gate delays and wire delays must be known. The input vectors for this timing simulation are just the vectors corresponding to the fanin and fanout states of i^* and j^* on each path. Notice also that we must simulate, except for fanout load effects on the delay, only the part of the circuit corresponding (either directly or through algebraic factorization) to the implicant of F whose delay we are measuring. Otherwise the measurement could be invalidated by a transition on the output of some other implicant.

The lower bound on d_3 can either be obtained in the same way (if we are also synthesizing the circuit for signal i) or from any other information source, such as a data sheet. In the worst case we can assume it to be zero.

Whenever Procedure 4.3 finds a hazard condition, we record the triple of paths and delays.

At the end, we have two choices for each triple (d_1, d_2, d_3) that failed the above test:

1. speed up d_2 and/or slow down d_1. This brings the circuit closer to the gate-delay case, where all delays inside the sub-circuit for each output t are supposed to be balanced. This may not always be possible, furthermore it may introduce cyclic dependencies, so that when we try to remove one hazard we can make another one worse.

2. slow down d_3. This is always possible (just add buffers after the output of the sub-circuit for signal i). Furthermore we are guaranteed that *we do not introduce cyclic constraints*, since d_1 and d_2 are measured from a sub-circuit input to a flip-flop input, so adding delay *after* a flip-flop does not change any of them.

If we take the second choice, then we just record by how much each flip-flop output fails to pass the test, and then slow it down by the maximum difference.

Notice the close similarity between what we do here and what is classically done in *synchronous circuit* synthesis:

- in the *synchronous* case we *slow down the clock signal until no more events are propagating along the whole circuit.*

- in the *asynchronous* case we *slow down each signal until no more events that caused its change are propagating along the whole circuit.*

So this approach, even if it does not generate speed-independent or delay-insensitive circuits, can still be considered faithfully adherent to the "asynchronous philosophy", in that every element must obey a "locally defined" protocol, and elements that are logically far apart must not be slowed down due to each other.

If we are synthesizing also the circuit for signal i, a specially bad case for Procedure 4.3 would be $d_3 = 0$, that corresponds to signal i being identically equal to signal j. But signal i can be equal to signal j, given Procedure 4.2, if and only if every transition for j is immediately followed by the same transition for i. Such condition can be easily detected on the STG, and we can just collapse i^* nodes into j^* nodes without changing the specified behavior. In all other cases, the delay between j^* and i^* is at least one logic gate.

4.6 STG Persistency and hazards

An STG is *persistent* ([Chu87]) if all its transitions are persistent. A transition u^* is persistent if for each immediate predecessor t^* of u^*, u^* and $\overline{t^*}$ are ordered.

For example in Figure 2.a transition y^+ is not persistent, because it has x^+ as a predecessor, but x^- and y^+ are concurrent (that is not ordered).

In a persistent STG whenever a transition u^* becomes enabled, none of its enabling signals t can change level before u^* has fired.

A transition t^* *disables* a transition u^* if there exists an STG marking m where both t^* and u^* are enabled, but firing t^* brings to a marking m' where u^* is no longer enabled. Notice that the only transitions that can be disabled in an FC net are FC

d_3 can be measured on the circuit for output x applying transition z^+, that is from vertex $x\overline{y}\overline{z}$ to $x\overline{y}\overline{z}$.
Again d_3 is the reset delay of an S-R flip-flop, so it is generally going to be larger than the difference between the switching times of two inputs of a *nor* gate. So we do not have to slow down the circuit.

2. valid vector pair $(110, 011)$, with transition cube $-1-$: same as the previous case.

All other transition cubes are either covered by implicants or do not intersect them at all.

z : All transition cubes are either covered by implicants or do not intersect them at all.